

From Regexes to Parsing Expression Grammars

Sérgio Medeiros

Department of Computer Science – UFS – Aracaju – Brazil

Fabio Mascarenhas

Department of Computer Science – UFRJ – Rio de Janeiro – Brazil

Roberto Ierusalimschy

Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil

Abstract

Most scripting languages nowadays use regex pattern-matching libraries. These regex libraries borrow the syntax of regular expressions, but have an informal semantics that is different from the semantics of regular expressions, removing the commutativity of alternation and adding ad-hoc extensions that cannot be expressed by formalisms for efficient recognition of regular languages, such as deterministic finite automata.

Parsing Expression Grammars are a formalism that can describe all deterministic context-free languages and has a simple computational model. In this paper, we present a formalization of regexes via transformation to Parsing Expression Grammars. The proposed transformation easily accommodates several of the common regex extensions, giving a formal meaning to them. It also provides a clear computational model that helps to estimate the efficiency of regex-based matchers, and a basis for specifying provably correct optimizations for them.

Keywords: regular expressions, parsing expression grammars, natural semantics, pattern matching, regexes

Email addresses: `sergio@ufs.br` (Sérgio Medeiros), `mascarenhas@ufrj.br` (Fabio Mascarenhas), `roberto@inf.puc-rio.br` (Roberto Ierusalimschy)

1. Introduction

Regular expressions are a concise way for describing regular languages with an algebraic notation. Their syntax has seen wide use in pattern matching libraries for programming languages, where they are used to specify the pattern against which a user is trying to match a string, or, more commonly, the pattern that a user is searching for in a string.

Regular expressions used for pattern matching are known as *regexes* [1, 2], and while they look like regular expressions they often have different semantics, based on how the pattern matching libraries that use them are actually implemented.

A simple example that shows this semantic difference are the regular expressions $a|aa$ and $aa|a$, which both describe the language $\{a, aa\}$. It is trivial to prove that the $|$ operator of regular expressions is commutative given its common semantics as the union of sets. But the *regexes* $a|aa$ and $aa|a$ behave differently for some implementations of pattern matching libraries and some subjects.

The standard regex libraries of the Perl and Ruby languages, as well as PCRE [3], a regex library with bindings for many programming languages, give different results when matching these two regexes against the subject **aa**. In all three libraries the first regex matches just the first **a** of the subject, while the second regex matches the whole subject. With the subject **ab** both regexes give the same answer in all three libraries, matching the first **a**, but we can see that the $|$ operator for regexes is not commutative.

This behavior of regexes is directly linked to the way they are usually implemented, by trying the alternatives in a $|$ expression in the order they appear and backtracking when a particular path through the expression makes the match fail.

A naive implementation of regex matching via backtracking can have exponential worst-case running time, which implementations try to avoid through ad-hoc optimizations to cut the amount of backtracking that needs to be done for common patterns. These ad-hoc optimizations lead to implementations not having a cost model of their operation, which makes it difficult for users to determine the performance of regex patterns. Simple modifications can make the time complexity of a pattern go from linear to exponential in unpredictable ways [4, 5].

Regexes can also have syntactical and semantical extensions that are difficult, or even impossible, to express through pure regular expressions. These

extensions do not have a formal model, but are informally specified through how they modify the behavior of an implementation based on backtracking. The meaning of regex patterns that use the extensions may vary among different regex libraries [6], or even among different implementations of the same regex library [7].

Practical regex libraries try to solve performance problems with ad-hoc optimizations for common patterns, but this makes the implementation of a regex library a complex task, and is another source of unpredictable performance, as different implementations can have different performance characteristics.

A heavily optimized regex engine, RE2 [8], uses an implementation based on finite automata and guarantees linear time performance, but it relies on ad-hoc optimizations to handle more complex patterns, as a naive automata-based implementation can have quadratic behavior [9]. More importantly, it cannot implement some common regex extensions [8].

Parsing Expression Grammars (PEGs) [10] are a formalism that can express all deterministic context-free languages, which means that PEGs can also express all regular languages. The syntax of PEGs is based on the syntax of regular expressions and regexes, and PEGs have a formal semantics based on *ordered choice*, a controlled form of backtracking that, like the `|` operation of regexes, is sensitive to the ordering of the alternatives.

We believe that ordered choice makes PEGs a suitable base for a formal treatment of regexes, and show, in this paper, that we can describe the meaning of regex patterns by conversion to PEGs. Moreover, PEGs can be efficiently executed by a parsing machine that has a clear cost model that we can use to reason about the time complexity of matching a given pattern [11, 12]. We can then use the semantics of PEGs to reason about the behavior of regexes, for example, to optimize pattern matching and searching by avoiding excessive backtracking. We believe that the combination of the regex to PEG conversion and the PEG parsing machine can be used to build implementations of regex libraries that are simpler and easier to extend than current ones.

The main contribution of this paper is our formalization of a simple, structure-preserving translation from plain regular expressions to PEGs that can be used to translate regexes to PEGs that match the same subjects. We present a formalization of regular expressions as patterns that match prefixes of strings instead of sets of strings, using the framework of natural semantics [13]. In this semantics, regular expressions are just a non-deterministic

form of the regexes used for pattern matching. We show that our semantics is equivalent to the standard set-based semantics when we consider the language of a pattern as the set of prefixes that it matches.

We then present a formalization of PEGs in the same style, and use it to show the similarities and differences between regular expressions, regexes, and PEGs. We then define a transformation that converts a regular expression to a PEG, and prove its correctness. We also show how we can improve the transformation for some classes of regexes by exploiting their properties and the greater predictability and control of performance that PEGs have, improving the performance of the resulting PEGs. Finally, we show how our transformation can be adapted to accommodate four regex extensions that cannot be expressed by regular expressions: independent expressions, possessive and lazy repetition, and lookahead.

There are procedures for transforming deterministic finite automata and right-linear grammars to PEGs [11, 14] and, as there are transformations from regular expressions to these formalisms, we could have used these existing procedures as the basis of an implementation of regular expressions in PEG engines. But the transformations of regular expressions to these formalisms cover just a subset of regexes, not including common extensions, including those covered in Section 6 of this paper. The direct transformation we present here is straightforward and can cover regex extensions.

In the next section, we present our formalizations of regular expressions and PEGs, and discuss when a regular expression has the same meaning when interpreted as a regular expression and as a PEG, along with the intuition behind our transformation. In Section 3, we formalize our transformation from regular expressions to PEGs and prove its correctness. In Section 4 we show how we can reason about the performance of PEGs to improve the PEGs generated by our transformation in some cases. In Section 5 we show how our approach compares to existing regex implementations with some benchmarks. In Section 6 we show how our transformation can accommodate some regex extensions. Finally, in Section 7 we discuss some related work and present our conclusions.

2. Regular Expressions and PEGs

Given a finite alphabet T , we can define a regular expression e inductively as follows, where $a \in T$, and both e_1 and e_2 are also regular expressions:

$$e = \varepsilon \mid a \mid e_1 e_2 \mid e_1 \mid e_2 \mid e_1^*$$

Traditionally, a regular expression can also be \emptyset , but we will not consider it; \emptyset is not used in regexes, and any expression with \emptyset as a subexpression can be either rewritten without \emptyset or is equal to \emptyset .

Note that this definition gives an abstract syntax for expressions, and this abstract syntax is what we use in the formal semantics and proofs. In our examples we use a concrete syntax that assumes that juxtaposition (concatenation) is left-associative and has higher precedence than $|$, which is also left-associative, while $*$ has the highest precedence, and we will use parentheses for grouping when these precedence and associativity rules get in the way.

The language of a regular expression e , $L(e)$, is traditionally defined through operations on sets. Intuitively, the languages of ε and a are singleton sets with the corresponding symbols, the language of $e_1 e_2$ is given by concatenating all strings of $L(e_1)$ with all strings of $L(e_2)$, the language of $e_1 | e_2$ is the union of the languages of e_1 and e_2 , and the language of e_1^* is the Kleene closure of the language of e_1 , that is, $L^* = \bigcup_{i=0}^{\infty} L^i$ where $L^0 = \{\varepsilon\}$ and $L^i = LL^{i-1}$ for $i > 0$ [15, p. 28].

We are interested in a semantics for regexes, the kind of regular expressions used for pattern matching and searching, so we will define a *matching* relation for regular expressions, $\overset{\text{RE}}{\rightsquigarrow}$. Informally, we will have $e \ xy \overset{\text{RE}}{\rightsquigarrow} y$ if and only if the expression e matches the prefix x of input string xy .

Formally, we define $\overset{\text{RE}}{\rightsquigarrow}$ via natural semantics, using the set of inference rules in Figure 1. We have $e \ xy \overset{\text{RE}}{\rightsquigarrow} y$ if and only if we can build a proof tree for this statement using the inference rules. The rules follow naturally from the expected behavior of each expression: rule **empty.1** says that ε matches itself and does not consume the input; rule **char.1** says that a symbol matches and consumes itself if it is the beginning of the input; rule **con.1** says that a concatenation uses the suffix of the first match as the input for the next; rules **choice.1** and **choice.2** say that a choice can match the input using either option; finally, rules **rep.1** and **rep.2** say that a repetition can either match ε and not consume the input or match its subexpression and match the repetition again on the suffix that the subexpression left.

The following lemma proves that the set of strings that expression e matches is the language of e , that is, $L(e) = \{x \in T^* \mid \exists y \ e \ xy \overset{\text{RE}}{\rightsquigarrow} y, y \in T^*\}$.

Lemma 1. *Given a regular expression e and a string x , for any string y we have that $x \in L(e)$ if and only if $e \ xy \overset{\text{RE}}{\rightsquigarrow} y$.*

Proof. (\Rightarrow): By induction on the complexity of the pair (e, x) . Given the

$$\begin{array}{ll}
\textbf{Empty String} & \frac{}{\varepsilon \ x \overset{\text{RE}}{\rightsquigarrow} x} \textbf{(empty.1)} \qquad \textbf{Character} \quad \frac{}{a \ ax \overset{\text{RE}}{\rightsquigarrow} x} \textbf{(char.1)} \\
\\
\textbf{Concatenation} & \frac{e_1 \ xyz \overset{\text{RE}}{\rightsquigarrow} yz \quad e_2 \ yz \overset{\text{RE}}{\rightsquigarrow} z}{e_1 e_2 \ xyz \overset{\text{RE}}{\rightsquigarrow} z} \textbf{(con.1)} \\
\\
\textbf{Choice} & \frac{e_1 \ xy \overset{\text{RE}}{\rightsquigarrow} y}{e_1 \mid e_2 \ xy \overset{\text{RE}}{\rightsquigarrow} y} \textbf{(choice.1)} \qquad \frac{e_2 \ xy \overset{\text{RE}}{\rightsquigarrow} y}{e_1 \mid e_2 \ xy \overset{\text{RE}}{\rightsquigarrow} y} \textbf{(choice.2)} \\
\\
\textbf{Repetition} & \frac{}{e^* \ x \overset{\text{RE}}{\rightsquigarrow} x} \textbf{(rep.1)} \quad \frac{e \ xyz \overset{\text{RE}}{\rightsquigarrow} yz \quad e^* \ yz \overset{\text{RE}}{\rightsquigarrow} z}{e^* \ xyz \overset{\text{RE}}{\rightsquigarrow} z}, x \neq \varepsilon \textbf{(rep.2)}
\end{array}$$

Figure 1: Natural semantics of relation $\overset{\text{RE}}{\rightsquigarrow}$

pairs (e_1, x_1) and (e_2, x_2) , the first pair is more complex than the second one if and only if either e_2 is a proper subexpression of e_1 or $e_1 = e_2$ and $|x_1| > |x_2|$. The base cases are $(\varepsilon, \varepsilon)$ and (a, a) , and their proofs follow by application of rules **empty.1** and **char.1**, respectively. Cases $(e_1 e_2, x)$ and $(e_1 \mid e_2, x)$ use a straightforward application of the induction hypothesis on the subexpressions, followed by application of rule **con.1** or one of the **choice** rules. Case (e^*, ε) follows directly from rule **rep.1**, while for case (e^*, x) , where $x \neq \varepsilon$, we know by the definition of the Kleene closure that $x \in L^i(e_1)$ with $i > 0$, where $L^i(e_1)$ is $L(e_1)$ concatenated with itself i times. This means that we can decompose x into $x_1 x_2$, with a non-empty x_1 , where $x_1 \in L(e_1)$ and $x_2 \in L^{i-1}(e_1)$. Again by the definition of the Kleene closure this means that $x_2 \in L(e_1^*)$. The proof now follows by the induction hypothesis on (e_1, x_1) and (e_1^*, x_2) and an application of rule **rep.2**.

(\Leftarrow): By induction on the height of the proof tree for $e \ xy \overset{\text{RE}}{\rightsquigarrow} y$. Most cases are straightforward; the interesting case is when the proof tree concludes with rule **rep.2**. By the induction hypothesis we have that $x \in L(e_1)$ and $y \in L(e_1^*)$. By the definition of the Kleene closure we have that $y \in L^i(e_1)$, so $xy \in L^{i+1}(e_1)$ and, again by the Kleene closure, $xy \in L(e_1^*)$, which concludes the proof. \square

Salomaa [16] developed a complete axiom system for regular expressions,

where any valid equation involving regular expressions can be derived from the axioms. The axioms of system F_1 are:

$$e_1 \mid (e_2 \mid e_3) = (e_1 \mid e_2) \mid e_3 \quad (1)$$

$$e_1(e_2e_3) = (e_1e_2)e_3 \quad (2)$$

$$e_1 \mid e_2 = e_2 \mid e_1 \quad (3)$$

$$e_1(e_2 \mid e_3) = e_1e_2 \mid e_1e_3 \quad (4)$$

$$(e_1 \mid e_2)e_3 = e_1e_3 \mid e_2e_3 \quad (5)$$

$$e \mid e = e \quad (6)$$

$$\varepsilon e = e \quad (7)$$

$$\emptyset e = \emptyset \quad (8)$$

$$e \mid \emptyset = e \quad (9)$$

$$e^* = \varepsilon \mid e^*e \quad (10)$$

$$e^* = (\varepsilon \mid e)^* \quad (11)$$

Salomaa's regular expressions do not have the ε case; the original axioms use \emptyset^* , which has the same meaning, as the only possible proof trees for \emptyset^* use **rep.1**. The following lemma shows that these axioms are valid under our semantics for regular expressions, if we take $e_1 = e_2$ to mean that e_1 and e_2 match the same sets of strings.

Lemma 2. *For each of the axioms of F_1 , if l is the expression on the left side and r is the expression on the right side, we have that $l \text{ } xy \stackrel{RE}{\rightsquigarrow} y$ if and only if $r \text{ } xy \stackrel{RE}{\rightsquigarrow} y$.*

Proof. Trivially true for axiom 8, as there are no proof trees for either the left or right sides of this axiom. For axioms 1 to 7 and for axiom 9 it is straightforward to use the subtrees of the proof tree of one side to build a proof tree for the other side. We can prove the validity of axiom 11 by an straightforward induction on the height of the proof trees for each side.

For axiom 10, we need to prove the identity $a^*a = aa^*$, by induction on the heights of the proof trees. From this identity the left side of axiom 10 follows by taking the subtrees of **rep.2** and combining them with **con.1** into a tree for aa^* , which means we have a tree for a^*a that we can use to build a tree for the right side using **choice.2**. The right side follows from getting a tree for aa^* from a tree for a^*a using the identity, then taking its subtrees and using **rep.2** to get a tree for a^* . \square

Parsing expression grammars borrow the syntax of regular expressions. A *parsing expression* is also defined inductively, extending the inductive definition of regular expressions with two new cases, A for a non-terminal, and $!e$ for a *not-predicate* of expression e . A PEG G is a tuple (V, T, P, p_S) where V is the set of non-terminals, T is the alphabet (set of terminals), P is a function from V to parsing expressions, and p_S is the parsing expression that the PEG matches (its starting parsing expression). We will use the notation $G[p]$ for a grammar derived from G where p_S is replaced by p while keeping V , T , and P the same. We will refer to both regular expressions and parsing expressions as just expressions, letting context disambiguate between both kinds.

While the syntax of parsing expressions is similar to the syntax of regular expressions, the behavior of the choice and repetition operators is very different. Choice in PEGs is *ordered*; a PEG will only try to match the right side of a choice if the left side cannot match any prefix of the input. Repetition in PEGs is *possessive*; a repetition will always consume as much of the input as it can match, regardless of whether this leads to failure or a shorter match for the whole pattern¹. To formally define ordered choice and possessive repetition we also need a way to express that an expression does not match a prefix of the input, so we need to introduce **fail** as a possible outcome of a match.

Figure 2 gives the definition of $\overset{\text{PEG}}{\rightsquigarrow}$, the matching relation for PEGs. As with regular expressions, we say that $G \ xy \overset{\text{PEG}}{\rightsquigarrow} y$ to express that the grammar G matches the prefix x of input string xy , and the set of strings that a PEG matches is its language: $L(G) = \{x \in T^* \mid \exists y \ G \ xy \overset{\text{PEG}}{\rightsquigarrow} y, y \in T^*\}$.

We mark with a $*$ the rules that have been changed from Figure 1, and mark with a $+$ the rules that were added. Unmarked rules are unchanged from Figure 1, except for the trivial change of adding the parameter G to the relation. We have six new rules, and two changed rules. New rules **char.2** and **char.3** generate **fail** in the case that the expression cannot match the symbol in the beginning of the input. New rule **con.2** says that a concatenation fails if its left side fails. New rule **var.1** says that to match a non-terminal we have to match the parsing expression associated with it in this grammar's function from non-terminals to parsing expressions (the non-

¹Possessive repetition is a consequence of ordered choice, as e^* is the same as expression A where A is a fresh non-terminal and $P(A) = eA \mid \varepsilon$.

$$\begin{array}{l}
\textbf{Empty String} \quad \frac{}{G[\varepsilon] \ x \xrightarrow{\text{PEG}} x} \textbf{(empty.1)} \quad \textbf{Non-terminal} \quad \frac{G[P(A)] \ x \xrightarrow{\text{PEG}} X}{G[A] \ x \xrightarrow{\text{PEG}} X} \textbf{(var.1)}^+ \\
\\
\textbf{Terminal} \quad \frac{}{G[a] \ ax \xrightarrow{\text{PEG}} x} \textbf{(char.1)} \quad \frac{}{G[b] \ ax \xrightarrow{\text{PEG}} \text{fail}} , b \neq a \textbf{(char.2)}^+ \quad \frac{}{G[a] \ \varepsilon \xrightarrow{\text{PEG}} \text{fail}} \textbf{(char.3)}^+ \\
\\
\textbf{Concatenation} \quad \frac{G[p_1] \ xy \xrightarrow{\text{PEG}} y \quad G[p_2] \ y \xrightarrow{\text{PEG}} X}{G[p_1 p_2] \ xy \xrightarrow{\text{PEG}} X} \textbf{(con.1)} \quad \frac{G[p_1] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[p_1 p_2] \ x \xrightarrow{\text{PEG}} \text{fail}} \textbf{(con.2)}^+ \\
\\
\textbf{Ordered Choice} \quad \frac{G[p_1] \ xy \xrightarrow{\text{PEG}} y}{G[p_1 \mid p_2] \ xy \xrightarrow{\text{PEG}} y} \textbf{(choice.1)} \quad \frac{G[p_1] \ x \xrightarrow{\text{PEG}} \text{fail} \quad G[p_2] \ x \xrightarrow{\text{PEG}} X}{G[p_1 \mid p_2] \ x \xrightarrow{\text{PEG}} X} \textbf{(choice.2)}^* \\
\\
\textbf{Repetition} \quad \frac{G[p] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[p^*] \ x \xrightarrow{\text{PEG}} x} \textbf{(rep.1)}^* \quad \frac{G[p] \ xyz \xrightarrow{\text{PEG}} yz \quad G[p^*] \ yz \xrightarrow{\text{PEG}} z}{G[p^*] \ xyz \xrightarrow{\text{PEG}} z} \textbf{(rep.2)} \\
\\
\textbf{Not Predicate} \quad \frac{G[p] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[!p] \ x \xrightarrow{\text{PEG}} x} \textbf{(not.1)}^+ \quad \frac{G[p] \ xy \xrightarrow{\text{PEG}} y}{G[!p] \ xy \xrightarrow{\text{PEG}} \text{fail}} \textbf{(not.2)}^+
\end{array}$$

Figure 2: Definition of Relation $\xrightarrow{\text{PEG}}$ through Natural Semantics

terminal's “right-hand side” in the grammar). New rules **not.1** and **not.2** say that a not predicate never consumes input, but fails if its subexpression matches a prefix of the input.

The change in rule **con.1** is trivial and only serves to propagate **fail**, so we do not consider it an actual change. The changes to rules **choice.2** and **rep.1** are what actually implements ordered choice and possessive repetition, respectively. Rule **choice.2** says that we can only match the right side of the choice if the left side fails, while rule **rep.1** says that a repetition only stops if we try to match its subexpression and fail.

It is easy to see that PEGs are deterministic; that is, a given PEG G can only have a single result (either **fail** or a suffix of x) for some input x , and only a single proof tree for this result. If the PEG G always yields a result for any input in T^* then we say that G is *complete* [10]. PEGs that are not complete include any PEG that has left recursion and PEGs with repetitions e^* where e matches the empty string. From now on we will assume that any PEG we consider is complete unless stated otherwise. The completeness of

a PEG can be proved syntactically [10].

The syntax of the expressions that form a PEG are a superset of the syntax of regular expressions, so syntactically any regular expression e has a corresponding PEG $G_e = (V, T, P, e)$, where V and P can be anything. We can prove that $L(G_e) \subseteq L(e)$ by a simple induction on the height of proof trees for G_e $xy \xrightarrow{\text{PEG}} y$, but it is easy to show examples where $L(G_e)$ is a proper subset of $L(e)$, so the regular expression and its corresponding PEG have different languages.

For example, expression $a|ab$ has the language $\{a, ab\}$ as a regular expression but $\{a\}$ as a PEG, because on an input with prefix ab the left side of the choice always matches and keeps the right side from being tried. The same happens with expression $a(b|bb)$, which has language $\{ab, abb\}$ as a regular expression and $\{ab\}$ as a PEG, and on inputs with prefix abb the left side of the choice keeps the right side from matching.

The behavior of the PEGs actually match the behavior of the *regexes* $a|ab$ and $a(b|bb)$ on Perl-compatible regex engines. These engines will always match $a|ab$ with just the first a on subjects starting with ab , and always match $a(b|bb)$ with ab on subjects starting with abb , unless the order of the alternatives is reversed.

A different situation happens with expression $(a|aa)b$. As a regular expression, its language is $\{ab, aab\}$ while as a PEG it is $\{ab\}$, but the PEG fails on an input with prefix aab . Regex engines will backtrack and try the second alternative when b fails to match the second a , and will also match aab , highlighting the difference between the unrestricted backtracking of common regex implementations and PEGs' restricted backtracking.

If we take the previous regular expression, $(a|aa)b$, and distribute b over the two alternatives, we have $ab|aab$. This expression now has the same language when we interpret it either as a regular expression, as a PEG, or as a regex.

If we change $a|ab$ and $a(b|bb)$ so they have the *prefix property*², by adding an end-of-input marker $\$$, we have the expressions $(a|ab)\$$ and $a(b|bb)\$$. Now their languages as regular expressions are $\{a\$, ab\ \$\}$ and $\{ab\$, abb\ \$\}$, respectively, but the first expression fails as a PEG on an input with prefix ab and the second expression fails as a PEG on an input with prefix abb .

Both $(a|ab)\$$ and $a(b|bb)\$$ match, as regexes, the same set of strings that

²There are no distinct strings x and y in the language such that x is a prefix of y .

form their languages as regular expressions. They are in the form $(e_1 \mid e_2) e_3$, like $(a \mid aa) b$, so we can distribute e_3 over the choice to obtain $e_1 e_3 \mid e_2 e_3$. If we do that the two expressions become $a\$ \mid ab\$$ and $a (b\$ \mid bb\$)$, respectively, and they now have the same language either as a regular expression, as a PEG, or as a regex.

We will say that a PEG G and a regular expression e over the same alphabet T are *equivalent* if the following conditions hold for every input string xy :

$$G \ xy \xrightarrow{\text{PEG}} y \Rightarrow e \ xy \xrightarrow{\text{RE}} y \quad (12)$$

$$e \ xy \xrightarrow{\text{RE}} y \Rightarrow G \ xy \not\xrightarrow{\text{PEG}} \text{fail} \quad (13)$$

That is, a PEG G and a regular expression e are equivalent if $L(G) \subseteq L(e)$ and G does not fail for any string with a prefix in $L(e)$. In the examples above, regular expressions $a \mid ab$, $a (b \mid bb)$, $a\$ \mid ab\$$, $a (b\$ \mid bb\$)$, and $ab \mid aab$ are all equivalent with their corresponding PEGs, while $(a \mid ab) \$$, $a (b \mid bb) \$$, and $(a \mid aa) b$ are not.

Informally, a PEG and a regular expression will be equivalent if the PEG matches the same strings as the regular expression when the regular expression is viewed as a regex under the common “leftmost-first” semantics of Perl-compatible regex implementations. If a regular expression can match different prefixes of the same subject, due to the non-determinism of the choice and repetition operations, the two conditions of equivalence guarantee that an equivalent PEG will match one of those prefixes.

Regexes are deterministic, and will also match just a single prefix of the possible prefixes a regular expression can match. Our transformation will preserve the ordering among choices, so the prefix an equivalent PEG obtained with our transformation matches will be the same prefix a regex matches.

While equivalence is enough to guarantee that a PEG will give the same results as a regex, equivalence together with the prefix property yields the following lemma:

Lemma 3. *If a regular expression e with the prefix property and a PEG G are equivalent then $L(G) = L(e)$.*

Proof. As our first condition of equivalence says that $L(G) \subseteq L(e)$, we just need to prove that $L(e) \subseteq L(G)$. Suppose there is a string $x \in L(e)$; this means that $e \ xy \xrightarrow{\text{RE}} y$ for any y . But from equivalence this means that

$G \ xy \not\stackrel{\text{PEG}}{\sim} \text{fail}$. As G is complete, we have $G \ xy \stackrel{\text{PEG}}{\sim} y'$. By equivalence, the prefix of xy that G matches is in $L(e)$. Now y cannot be a proper suffix of y' nor y' a proper suffix of y , or the prefix property would be violated. This means that $y' = y$, and $x \in L(G)$, completing the proof. \square

We can now present an overview on how we will transform a regular expression e into an equivalent PEG. We first need to transform subexpressions of the form $(e_1 \mid e_2) e_3$ to $e_1 e_3 \mid e_2 e_3$. We do not need to actually duplicate e_3 in both sides of the choice, potentially causing an explosion in the size of the resulting expression, but can introduce a fresh non-terminal X with $P(X) = e_3$, and distribute X to transform $(e_1 \mid e_2) e_3$ into $e_1 X \mid e_2 X$.

Transforming repetition is trickier, but we just have to remember that $e_1^* e_2 \equiv (e_1 e_1^* \mid \varepsilon) e_2 \equiv (e_1 e_1^* e_2) \mid e_2$. Naively transforming the first expression to the third does not work, as we end up with $e_1^* e_2$ in the expression again, but we can add a fresh non-terminal A to the PEG with $P(A) = e_1 A \mid e_2$ and then replace $e_1^* e_2$ with A in the original expression. The end result of repeatedly applying these two transformation steps until we reach a fixed point will be a parsing expression that is equivalent to the original regular expression.

As an example, let us consider the regular expression $b^* b \$$. Its language is $\{b \$, bb \$, \dots\}$, but when interpreted as a PEG the language is \emptyset , due to possessive repetition. If we transform the original expression into a PEG with starting parsing expression A and $P(A) = b A \mid b \$$, it will have the same language as the original regular expression; for example, given the input $bb \$$, this PEG matches the first b through subexpression b of $b A$, and then A tries to match the rest of the input, $b \$$. So, once more subexpression b of $b A$ matches b and then A tries to match the rest of the input, $\$$. Since both $b A$ and $b \$$ fail to match $\$$, A fails, and thus $b A$ fails for input $b \$$. Now we try $b \$$, which successfully matches $b \$$, and the complete match succeeds.

If we now consider the regular expression $b^* b$, which has $\{b, bb, \dots\}$ as its language, we have that it also has the empty set as its language if we interpret it as a PEG. A PEG with starting parsing expression A and $P(A) = b A \mid b$ also has $\{b, bb, \dots\}$ as its language, but with an important difference: the regular expression can match just the first b of a subject starting with bb , but the PEG will match both, and any other ones that follow, so we do not have $e \ xy \stackrel{\text{RE}}{\sim} y$ implying $G \ xy \stackrel{\text{PEG}}{\sim} y$. But the behavior of the PEG corresponds the behavior of regex engines, which use *greedy* repetition, where a repetition will always match as much as it can while not making the rest of the expression

$$\begin{aligned}
\Pi(\varepsilon, G_k) &= G_k & \Pi(a, G_k) &= G_k[a p_k] & \Pi(e_1 e_2, G_k) &= \Pi(e_1, \Pi(e_2, G_k)) \\
\Pi(e_1 \mid e_2, G_k) &= G_2[p_1 \mid p_2], \text{ where } G_2 = (V_2, T, P_2, p_2) = \Pi(e_2, (V_1, T, P_1, p_k)) \\
&\quad \text{and } (V_1, T, P_1, p_1) = \Pi(e_1, G_k) \\
\Pi(e_1^*, G_k) &= G, \text{ where } G = (V_1, T, P_1 \cup \{A \rightarrow p_1 \mid p_k\}, A) \text{ with } A \notin V_k \text{ and} \\
&\quad (V_1, T, P_1, p_1) = \Pi(e_1, (V_k \cup \{A\}, T, P_k, A))
\end{aligned}$$

Figure 3: Definition of Function Π , where $G_k = (V_k, T, P_k, p_k)$

fail.

The next section formalizes our transformation, and proves that for any regular expression e it will give a PEG that is equivalent to e , that is, it will yield a PEG that recognizes the same language as e if it has the prefix property.

3. Transforming Regular Expressions to PEGs

This section presents function Π , a formalization of the transformation we outlined in the previous section. The function Π transforms a regular expression e using a PEG G_k that is equivalent to a regular expression e_k to yield a PEG that is equivalent to the regular expression $e e_k$.

The intuition behind Π is that G_k is a *continuation* for the regular expression e , being what should be matched after matching e . We use this continuation when transforming choices and repetitions to do the transformations of the previous section; for a choice, the continuation is distributed to both sides of the choice. For a repetition, it is used as the right side for the new non-terminal, and the left side of this non-terminal is the transformation of the repetition's subexpression with the non-terminal as continuation.

For a concatenation, the transformation is the result of transforming the right side with G_k as continuation, then using this as continuation for transforming the left side. This lets the transformation of $(e_1 \mid e_2) e_3$ work as expected: we transform e_3 and then use the PEG as the continuation that we distribute over the choice.

We can transform a standalone regular expression e by passing a PEG with ε as starting expression as the continuation; this gives us a PEG that is equivalent to the regular expression $e \varepsilon$, or e .

Figure 3 has the definition of function Π . Notice how repetition introduces a new non-terminal, and the transformation of choice has to take this into account by using the set of non-terminals and the productions of the result of transforming one side to transform the other side, so there will be no overlap. Also notice how we transform a repetition by transforming its body using the repetition itself as a continuation (through the introduction of a fresh non-terminal), then building a choice between the transformation and the body and the continuation of the repetition. The transformation process is bottom-up and right-to-left.

We will show the subtler points of transformation Π with some examples. In the following discussion, we use the alphabet $T = \{a, b, c\}$, and the continuation grammar $G_k = (\emptyset, T, \emptyset, \varepsilon)$ that is equivalent to the regular expression ε . In our first example, we use the regular expression $(a | b | c)^* a (a | b | c)^*$, which matches an input that has at least one **a**.

We first transform the second repetition by evaluating $\Pi((a | b | c)^*, G_k)$; we first transform $a | b | c$ with a new non-terminal A as continuation, yielding the PEG $aA | bA | cA$, then combine it with ε to yield the PEG A where A has the production below:

$$A \rightarrow aA \mid bA \mid cA \mid \varepsilon$$

Next is the concatenation with a , yielding the PEG aA . We then use this PEG as continuation for transforming the first repetition. This transformation uses a new non-terminal B as a continuation for transforming $a | b | c$, yielding $aB | bB | cB$, then combines it with aA to yield the PEG B with the productions below:

$$B \rightarrow aB \mid bB \mid cB \mid aA \quad A \rightarrow aA \mid bA \mid cA \mid \varepsilon$$

When the original regular expression matches a given input, we do not know how many **a**'s the first repetition matches, because the semantics of regular expressions is non-deterministic. Regex implementations commonly resolve ambiguities in repetitions by the longest match rule, where the first repetition will match all but the last **a** of the input. PEGs are deterministic by construction, and the PEG generated by Π obeys the longest match rule. The alternative aA of non-terminal B will only be tried if all the alternatives fail, which happens in the end of the input. The PEG then backtracks until

the last **a** is found, where it matches the last **a** and proceeds with non-terminal A .

The regular expression $(b|c)^* a (a|b|c)^*$ defines the same language as the regular expression of the first example, but without the ambiguity. Now Π with continuation G_k yields the following PEG B :

$$B \rightarrow bB \mid cB \mid aA \quad A \rightarrow aA \mid bA \mid cA \mid \varepsilon$$

Although the productions of this PEG and the previous one match the same strings, the second PEG is more efficient, as it will not have to reach the end of the input and then backtrack until finding the last **a**. This is an example on how we can use our semantics and the transformation Π to reason about the behavior of a regex. The relative efficiency of the two PEGs is an artifact of the semantics, while the relative efficiency of the two regexes depends on how a particular engine is implemented. In a backtracking implementation it will depend on what ad-hoc optimizations the implementation makes, in an automata implementation they both will have the same relative efficiency, at the expense of the implementation lacking the expressive power of some regex extensions.

The expressions in the two previous examples are *well-formed*. A regular expression e is well-formed if it does not have a subexpression e_i^* where $\varepsilon \in L(e_i)$. If e is a well-formed regular expression and G_k is a complete PEG then $\Pi(e, G_k)$ is also complete. In Section 3.1 we will show how to mechanically obtain a well-formed regular expression that recognizes the same language as a non-well-formed regular expression while preserving its overall structure.

We will now prove that our transformation Π is correct, that is, if e is a well-formed regular expression and G_k is a PEG equivalent to a regular expression e_k then $\Pi(e, G_k)$ is equivalent to $e e_k$. The proofs use a small technical lemma: each production of PEG G_k is also in PEG $\Pi(e, G_k)$, for any regular expression e . This lemma is straightforward to prove by structural induction on e .

We will prove each property necessary for equivalence separately; equivalence will then be a direct corollary of those two proofs. To prove the first property we need an auxiliary lemma that states that the continuation grammar is indeed a continuation, that is if the PEG $\Pi(e, G_k)$ matches a prefix x of a given input xy then we can split x into v and w with $x = vw$ and G_k matching w .

Lemma 4. *Given a well-formed regular expression e , a PEG G_k , and an input string xy , if $\Pi(e, G_k) \ xy \xrightarrow{\text{PEG}} y$ then there is a suffix w of x such that $G_k \ wy \xrightarrow{\text{PEG}} y$.*

Proof. By induction on the complexity of the pair (e, xy) . The interesting case is e^* . In this case $\Pi(e^*, G_k)$ gives us a grammar $G = (V_1, T, P, A)$, where $A \rightarrow p_1 \mid p_k$. By **var.1** we know that $G[p_1 \mid p_k] \ xy \xrightarrow{\text{PEG}} y$. There are now two subcases to consider, **choice.1** and **choice.2**.

For subcase **choice.2**, we have $G[p_k] \ xy \xrightarrow{\text{PEG}} y$. But then we have that $G_k[p_k] \ xy \xrightarrow{\text{PEG}} y$ because any non-terminal that p_k uses to match xy is in both G and G_k and has the same production in both. The string xy is a suffix of itself, and p_k is the starting expression of G_k , closing this part of the proof.

For subcase **choice.1** we have $\Pi(e, \Pi(e^*, G_k)) \ xy \xrightarrow{\text{PEG}} y$, and by the induction hypothesis $\Pi(e^*, G_k) \ wy \xrightarrow{\text{PEG}} y$. We can now use the induction hypothesis again, on the length of the input, as w must be a proper suffix of x . We conclude that $G_k \ w'y \xrightarrow{\text{PEG}} y$ for a suffix w' of w , and so a suffix of x , ending the proof. \square

The following lemma proves that if the first property of equivalence holds between a regular expression e_k and a PEG G_k then it will hold for $e e_k$ and $\Pi(e, G_k)$ given a regular expression e .

Lemma 5. *Given two well-formed regular expressions e and e_k and a PEG G_k , where $G_k \ wy \xrightarrow{\text{PEG}} y \Rightarrow e_k \ wy \xrightarrow{\text{RE}} y$, if $\Pi(e, G_k) \ vwy \xrightarrow{\text{PEG}} y$ then $e e_k \ vwy \xrightarrow{\text{RE}} y$.*

Proof. By induction on the complexity of the pair (e, vwy) . The interesting case is e^* . In this case, $\Pi(e^*, G_k)$ gives us a PEG $G = (V_1, T, P, A)$, where $A \rightarrow p_1 \mid p_k$. By **var.1** we know that $G[p_1 \mid p_k] \ vwy \xrightarrow{\text{PEG}} y$. There are now two subcases, **choice.1** and **choice.2** of $\xrightarrow{\text{PEG}}$.

For subcase **choice.2**, we can conclude that $G_k \ vwy \xrightarrow{\text{PEG}} y$ because p_k is the starting expression of G_k and any non-terminals it uses have the same production both in G and G_k . We now have $e_k \ vwy \xrightarrow{\text{RE}} y$. By **choice.2** of $\xrightarrow{\text{RE}}$ we have $e e^* e_k \mid e_k \ vwy \xrightarrow{\text{RE}} y$, but $e e^* e_k \mid e_k \equiv e^* e_k$, so $e^* e_k \ vwy \xrightarrow{\text{RE}} y$, ending this part of the proof.

For subcase **choice.1**, we have $\Pi(e, \Pi(e^*, G_k)) \ vwy \xrightarrow{\text{PEG}} y$, and by Lemma 4 we have $\Pi(e^*, G_k) \ wy \xrightarrow{\text{PEG}} y$. The string v is not empty, so we can use the induction hypothesis and Lemma 4 again to conclude $e^* e_k \ wy \xrightarrow{\text{RE}} y$.

Then we use the induction hypothesis on $\Pi(e, \Pi(e^*, G_k)) \ vwy \xrightarrow{\text{PEG}} y$ to conclude $ee^*e_k \ vwy \xrightarrow{\text{RE}} y$. We can now use rule **choice.1** of $\xrightarrow{\text{RE}}$ to get $ee^*e_k \mid e_k \ vwy \xrightarrow{\text{RE}} y$, but $ee^*e_k \mid e_k \equiv e^*e_k$, so $e^*e_k \ vwy \xrightarrow{\text{RE}} y$, ending the proof. \square

The following lemma proves that if the second property of equivalence holds between a regular expression e_k and a PEG G_k then it will hold for ee_k and $\Pi(e, G_k)$ given a regular expression e .

Lemma 6. *Given well-formed regular expressions e and e_k and a PEG G_k , where Lemma 5 holds and we have $e_k \ wy \xrightarrow{\text{RE}} y \Rightarrow G_k \ wy \not\xrightarrow{\text{PEG}} \text{fail}$, if $ee_k \ vwy \xrightarrow{\text{RE}} y$ then $\Pi(e, G_k) \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$.*

Proof. By induction on the complexity of the pair (e, vwy) . The interesting case is e^* . We will use again the equivalence $e^*e_k \equiv ee^*e_k \mid e_k$. There are two subcases, **choice.1** and **choice.2** of $\xrightarrow{\text{RE}}$.

For subcase **choice.1**, we have that e matches a prefix of vwy by rule **con.1**. As e^* is well-formed this prefix is not empty, so $e^*e_k \ v'wy \xrightarrow{\text{RE}} y$ for a proper suffix v' of v . By the induction hypothesis we have $\Pi(e^*, G_k) \ v'wy \not\xrightarrow{\text{PEG}} \text{fail}$, and by induction hypothesis again we get $\Pi(e, \Pi(e^*, G_k)) \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$. This PEG is complete, so we can conclude $\Pi(e^*, G_k)[p_1 \mid p_k] \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$ using rule **choice.1** of $\xrightarrow{\text{PEG}}$, and then $\Pi(e^*, G_k) \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$ by rule **var.1**, ending this part of the proof.

For subcase **choice.2**, we can assume that there is no proof tree for the statement $ee^*e_k \ vwy \xrightarrow{\text{RE}} y$, or we could reduce this subcase to the first one by using **choice.1** instead of **choice.2**. Because $\Pi(e, \Pi(e^*, G_k))$ is complete we can use modus tollens of Lemma 5 to conclude that $\Pi(e, \Pi(e^*, G_k)) \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$. We also have $e_k \ vwy \xrightarrow{\text{RE}} y$, so $G_k \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$. Now we can use rule **choice.2** of $\xrightarrow{\text{PEG}}$ to conclude $G[p_1 \mid p_k] \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$, and then $\Pi(e^*, G_k) \ vwy \not\xrightarrow{\text{PEG}} \text{fail}$ by rule **var.1**, ending the proof. \square

The correctness lemma for Π is a corollary of the two previous lemmas:

Lemma 7. *Given well-formed regular expressions e and e_k and a PEG G_k , where e_k and G_k are equivalent, then $\Pi(e, G_k)$ and ee_k are equivalent.*

Proof. The proof that first property of equivalence holds for $\Pi(e, G_k)$ and ee_k follows from the first property of equivalence for e_k and G_k plus Lemma 5. The proof that the second property of equivalence holds follows from the

first property of equivalence for $\Pi(e, G_k)$ and $e e_k$, the second property of equivalence for e_k and G_k , plus Lemma 6. \square

A corollary of the previous lemma combined with Lemma 3 is that $L(e \$) = L(\Pi(e, \$))$, proving that our transformation can yield a PEG that recognizes the same language as any well-formed regular expression e just by using an end-of-input marker, even if the language of e does not have the prefix property.

It is interesting to see whether the axioms of system F_1 (presented on page 7) are still valid if we transform both sides using Π with ε as the continuation PEG, that is, if l is the left side of the equation and r is the right side then $\Pi(l, \varepsilon) \overset{\text{PEG}}{\rightsquigarrow} xy$ if and only if $\Pi(r, \varepsilon) \overset{\text{PEG}}{\rightsquigarrow} y$. This is straightforward for axioms 1, 2, 4, 6, 7, 8, and 9; in fact, it is easy to prove that these axioms will be valid for any PEG, not just PEGs obtained from our transformation.

Applying Π to both sides of axiom 2, $e_1(e_2e_3)$ and $(e_1e_2)e_3$, makes them identical; they both become $\Pi(e_1, \Pi(e_2, \Pi(e_3, G_k)))$. The same thing happens with axiom 5, $(e_1 | e_2)e_3 = e_1e_3 | e_2e_3$; the transformation of the left side, $\Pi((e_1 | e_2)e_3, G_k)$, becomes $\Pi(e_1, \Pi(e_3, G_k)) | \Pi(e_2, \Pi(e_3, G_k))$ via the intermediate expression $\Pi(e_1 | e_2, \Pi(e_3, G_k))$, while the transformation of the right side, $\Pi(e_1e_3 | e_2e_3, G_k)$, also becomes $\Pi(e_1, \Pi(e_3, G_k)) | \Pi(e_2, \Pi(e_3, G_k))$, although via the intermediate expression $\Pi(e_1e_3, G_k) | \Pi(e_2e_3, G_k)$.

The transformation of axiom 3, $e_1 | e_2 = e_2 | e_1$, will not be valid; the left side becomes the PEG $\Pi(e_1, G_k) | \Pi(e_2, G_k)$ and the right side becomes the PEG $\Pi(e_2, G_k) | \Pi(e_1, G_k)$, but ordered choice is not commutative in the general case. One case where this choice is commutative is if the language of $e_1 | e_2$ has the prefix property. We can use an argument analogous to the argument of Lemma 3 to prove this, which is not surprising, as this lemma together with Lemma 2 implies that this axiom should hold for expressions with languages that have the prefix property.

Axiom 10, $e^* = \varepsilon | e^*e$, needs to be rewritten as $e^* = e^*e | \varepsilon$ or it is trivially not valid, as the right side will always match just ε . Again, this is not surprising, as $\varepsilon | e^*e$ does not have the prefix property, and this is the same behavior of regex implementations. Rewriting the axiom as $e^* = e^*e | \varepsilon$ makes it valid when we apply Π to both sides, as long as e^* is well-formed.

The left side becomes the PEG A where $A \rightarrow \Pi(e, A) | \varepsilon$, while the right side becomes $B | \varepsilon$ where $B \rightarrow \Pi(e, B) | \Pi(e, \varepsilon)$. If $\Pi(e, A)$ fails it means that $\Pi(e, \varepsilon)$ would also fail, and so do $\Pi(e, B)$, then B fails and the $B | \varepsilon$ succeeds by **choice.2**. Analogous reasoning holds for the other side, if B fails. If

$\Pi(e, A)$ succeeds then $\Pi(e, \varepsilon)$ matches a non-empty prefix, and A matches the rest, and we can assume that $B \mid \varepsilon$ matches this rest by induction on the length of the matched string. We can use this to conclude that $B \mid \varepsilon$ also succeeds. Again, analogous reasoning holds for the converse.

The right side of axiom 11, $(\varepsilon \mid e)^*$, is not well-formed, and applying Π to it would lead to a left-recursive PEG with no possible proof tree. We still need to show that any regular expression can be made well-formed without changing its language. This is the topic of the next section, where we give a transformation that rewrites non-well-formed repetitions so they are well-formed with minimal changes to the structure of the original regular expression. Applying this transformation to the right side of axiom 11 will make it identical to the left side, making the axiom trivially valid.

3.1. Transformation of Repetitions e^* where $\varepsilon \in L(e)$

A regular expression e that has a subexpression e_i^* where e_i can match the empty string is not well-formed. As e_i can succeed without consuming any input one outcome of e_i^* is to stay in the same place of the input indefinitely. Regex libraries that rely on backtracking may enter an infinite loop with non-well-formed expressions unless they take measures to avoid it, using ad-hoc rules to detect and break the resulting infinite loops [17].

When e is not well-formed, the PEG we obtain through transformation Π is not complete. A PEG that is not complete can make a PEG library enter an infinite loop. To show an example on how a non-well-formed regular expression leads to a PEG that is not complete, let us transform $(a \mid \varepsilon)^* b$ using Π . Using ε as continuation yields the following PEG A :

$$A \rightarrow a A \mid A \mid b$$

The PEG above is *left recursive*, so it is not complete. In fact, this PEG does not have a proof tree for any input, so it is not equivalent to the regular expression $(a \mid \varepsilon)^* b$.

Transformation Π is not correct for non-well-formed regular expressions, but we can make any non-well-formed regular expression well-formed by rewriting repetitions e_i^* where $\varepsilon \in L(e_i)$ as $e_i'^*$ where $\varepsilon \notin L(e_i')$ and $L(e_i'^*) = L(e_i^*)$. The regular expression above would become $a^* \mid b$, which Π transforms into an equivalent complete PEG.

This section presents a transformation that mechanically performs this rewriting. We use a pair of functions to rewrite an expression, f_{out} and f_{in} .

$$\begin{aligned}
\text{empty}(\varepsilon) &= \mathbf{true} \\
\text{empty}(a) &= \mathbf{false} \\
\text{empty}(e^*) &= \text{empty}(e) \\
\text{empty}(e_1 e_2) &= \text{empty}(e_1) \wedge \text{empty}(e_2) \\
\text{empty}(e_1 \mid e_2) &= \text{empty}(e_1) \wedge \text{empty}(e_2)
\end{aligned}$$

$$\begin{aligned}
\text{null}(\varepsilon) &= \mathbf{true} \\
\text{null}(a) &= \mathbf{false} \\
\text{null}(e^*) &= \mathbf{true} \\
\text{null}(e_1 e_2) &= \text{null}(e_1) \wedge \text{null}(e_2) \\
\text{null}(e_1 \mid e_2) &= \text{null}(e_1) \vee \text{null}(e_2)
\end{aligned}$$

Figure 4: Definition of predicates *empty* and *null*

Function f_{out} recursively searches for a repetition that has ε in the language of its subexpression, while f_{in} rewrites the repetition's subexpression so it is well-formed, does not have ε in its language, and does not change the language of the repetition. Both f_{in} and f_{out} use two auxiliary predicates, *empty* and *null*, that respectively test if an expression is equal to ε (if its language is the singleton set $\{\varepsilon\}$) and if an expression has ε in its language. Figure 4 has inductive definitions for the *empty* and *null* predicates.

Function f_{out} is simple: for the base expressions it is the identity, for the composite expressions f_{out} applies itself recursively to subexpressions unless the expression is a repetition where the repetition's subexpression matches ε . In this case f_{out} transforms the repetition to ε if the subexpression is equal to ε (as $\varepsilon^* \equiv \varepsilon$), or uses f_{in} to rewrite the subexpression. Figure 5 has the inductive definition of f_{out} . It obeys the following lemma:

Lemma 8. *If $f_{in}(e_k)$ is well-formed, $\varepsilon \notin L(f_{in}(e_k))$, and $L(f_{in}(e_k)^*) = L(e_k^*)$ for any e_k with $\varepsilon \in L(e_k)$ and $L(e_k) \neq \varepsilon$ then, for any e , $f_{out}(e)$ is well-formed and $L(e) = L(f_{out}(e))$.*

Proof. By structural induction on e . Inductive cases follow directly from the induction hypothesis, except for e^* where $\varepsilon \in L(e)$, where it follows from the properties of f_{in} . \square

$$\begin{aligned}
f_{out}(e) &= e, \text{ if } e = \varepsilon \text{ or } e = a \\
f_{out}(e_1 e_2) &= f_{out}(e_1) f_{out}(e_2) \\
f_{out}(e_1 | e_2) &= f_{out}(e_1) | f_{out}(e_2) \\
f_{out}(e^*) &= \begin{cases} f_{out}(e)^* & \text{if } \neg null(e) \\ \varepsilon & \text{if } empty(e) \\ f_{in}(e)^* & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 5: Definition of Function f_{out}

$$\begin{aligned}
f_{in}(e_1 e_2) &= f_{in}(e_1 | e_2) \\
f_{in}(e_1 | e_2) &= \begin{cases} f_{in}(e_2) & \text{if } empty(e_1) \text{ and } null(e_2) \\ f_{out}(e_2) & \text{if } empty(e_1) \text{ and } \neg null(e_2) \\ f_{in}(e_1) & \text{if } null(e_1) \text{ and } empty(e_2) \\ f_{out}(e_1) & \text{if } \neg null(e_1) \text{ and } empty(e_2) \\ f_{out}(e_1) | f_{in}(e_2) & \text{if } \neg null(e_1) \text{ and } \neg empty(e_2) \\ f_{in}(e_1) | f_{out}(e_2) & \text{if } \neg empty(e_1) \text{ and } \neg null(e_2) \\ f_{in}(e_1) | f_{in}(e_2) & \text{otherwise} \end{cases} \\
f_{in}(e^*) &= \begin{cases} f_{in}(e) & \text{if } null(e) \\ f_{out}(e) & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 6: Definition of Function $f_{in}(e)$, where $\neg empty(e)$ and $null(e)$

Function f_{in} does the heavy lifting of the rewriting, it is used when f_{out} finds an expression e^* where $\neg empty(e)$ and $null(e)$. If its argument is a repetition it throws away the repetition because it is superfluous. Then f_{in} applies f_{out} or itself to the subexpression depending on whether it matches ε or not. If the argument is a choice f_{in} throws away one of the sides if its equal to ε , as it is superfluous because of the repetition, and rewrites the remaining side using f_{out} or f_{in} depending on whether it matches ε or not. In case both sides are not equal to ε f_{in} rewrites both. If the argument is a concatenation f_{in} rewrites it as a choice and applies itself to the choice.

Transforming a concatenation into a choice obviously is not a valid transformation in the general case, but it is safe in the context of f_{in} ; f_{in} is working

inside a repetition expression, and its argument has ε in its language, so we can use an identity involving languages and the Kleene closure that says $(AB)^* = (A \cup B)^*$ if $\varepsilon \in A$ and $\varepsilon \in B$. Figure 6 has the inductive definition of f_{in} . It obeys the following lemma:

Lemma 9. *If $f_{out}(e_k)$ is well-formed and $L(f_{out}(e_k)) = L(e_k)$ for any e_k then, for any e with $\varepsilon \in L(e)$ and $L(e) \neq \{\varepsilon\}$, $\varepsilon \notin L(f_{in}(e))$, $L(e^*) = L(f_{in}(e)^*)$, and $f_{in}(e)$ is well-formed.*

Proof. By structural induction on e . Most cases follow directly from the induction hypothesis and the properties of f_{out} . The subcases of choice where the result is also a choice use the Kleene closure property $(A \cup B)^* = (A^* \cup B^*)^*$ together with the induction hypothesis and the properties of f_{out} . Concatenation becomes to a choice using the property mentioned above this lemma. \square

As an example, let us use f_{out} and f_{in} to rewrite the regular expression $(bc|a^*(d|\varepsilon))^*$ into a well-formed regular expression. We show the sequence of steps below:

$$\begin{aligned} f_{out}((bc|a^*(d|\varepsilon))^*) &= (f_{in}(bc|a^*(d|\varepsilon)))^* = (f_{out}(bc)|f_{in}(a^*(d|\varepsilon)))^* \\ &= (f_{out}(b)f_{out}(c)|(f_{in}(a^*)|f_{in}(d|\varepsilon)))^* \\ &= (bc|(f_{out}(a)|f_{out}(d)))^* = (bc|(a|d))^* \end{aligned}$$

The idea is for rewriting to be automated, and transparent to the user of regex libraries based on our transformation, unless the user wants to see how their expression can be simplified. Notice that just the presence of ε inside a repetition does not mean that a regular expression is not well-formed. The $(bc|a(d|\varepsilon))^*$ expression looks very similar to the previous one, but is well-formed and left unmodified by f_{out} .

4. Optimizing Search and Repetition

A common application of regexes is to search for parts of a subject that match some pattern, but our formal model of regular expressions and PEGs is *anchored*, as our matches must start on the first symbol of the subject instead of starting anywhere. It is easy to build a PEG that will search for another PEG (V, T, P, S) , though, we just need to add a new non-terminal S' as starting pattern, with $S' \rightarrow S|.S'$ as a new production, where $.$ is

a shortcut for a regular expression that matches any terminal. If trying to match S from the beginning of the subject fails, then the PEG skips the first symbol and tries again on the second.

The search pattern works, but can be very inefficient if the PEG engine always has to use backtracking to implement ordered choice, as advancing to the correct starting position may involve a large amount of advancing and then backtracking. A related problem occurs when converting regex repetition into PEGs, as the PEG generated from the regular expression $e_1^*e_2$ will greedily try to consume as much of the subject with e_1 as possible, then try e_2 and backtrack each match of e_1 until e_2 succeeds or the whole pattern fails. In the rest of this section we will show how we can use properties of the expressions we are trying to search or match in conjunction with syntactic predicates to reduce the amount of backtracking necessary in both searches and repetition expressions.

4.1. Search

The search pattern for a PEG tries to match the PEG then advances one position in the subject and tries again if the match fails. A simple way to improve this is to advance several positions if possible, skipping starting positions that have no chance of a successful match. If we know that a successful match always consumes part of the subject and begins with a symbol in a set \mathcal{F} then we can skip a failing starting position with the pattern $![\mathcal{F}]$, where $[\mathcal{F}]$ is a *character set* pattern that matches any symbol in the set. We can skip a string of failing symbols with the pattern $(![\mathcal{F}])^*$. The new search expression for the PEG with starting pattern p can be written as follows:

$$S \rightarrow ([\mathcal{F}])^*(p | .S)$$

The set \mathcal{F} for a pattern p derived from a regular expression e is just the *FIRST* set of e , which has a simple definition in terms of the $\overset{\text{RE}}{\rightsquigarrow}$ relation below:

$$\text{FIRST}(e) = \{a \in T \mid \exists x, y \ e \ axy \overset{\text{RE}}{\rightsquigarrow} y, \ x, y \in T^*\}$$

It is easy to prove that the two search expressions are equivalent by induction on the height of the corresponding derivation trees. The tricky case, where $(![\mathcal{F}])^*$, just uses the definition of *FIRST* to build a tree of successive applications of rule **ord.2** until we can use the induction hypothesis in its rightmost leaf.

4.2. Repetition

If two regular expressions e_1 and e_2 have disjoint *FIRST* sets then it is safe to match $e_1^*e_2$ using possessive repetition. This means that we can transform $e_1^*e_2$ into the PEG $p_1^*p_2$ where p_1 and p_2 are the PEGs we get from transforming e_1 and e_2 . Formally, we can define $\Pi(e_1^*e_2, G_k)$ when $FIRST(e_1) \cap FIRST(e_2) = \emptyset$ as follows, where we use $G_2[\varepsilon]$ as the continuation for transforming e_1 just to avoid collisions on the names of non-terminals:

$$\begin{aligned} \Pi(e_1^*e_2, G_k) &= (V_1, T, P_1, p_1^*p_2) \\ \text{where } (V_1, T, P_1, p_1) &= \Pi(e_1, G_2[\varepsilon]) \\ \text{and } G_2 &= (V_2, T, P_2, p_2) = \Pi(e_2, G_k) \end{aligned}$$

The easiest way to prove the correctness of the new rule is by proving the equivalence of the PEGs we get from $\Pi(e_1^*e_2, G_k)$ using the old and new rule. This is a straightforward induction on the height of the proof trees for these PEGs, using the fact that disjoint *FIRST* sets for e_1 and e_2 implies disjointedness of their equivalent PEGs.

In the general case, where the *FIRST* sets of e_1 and e_2 are not disjoint, we can still avoid some amount of backtracking on $e_1^*e_2$ by being possessive whenever there is no chance of e_2 doing a successful match, as backtracking to a point where e_2 cannot match is useless. The idea is to use a predicated repetition of p_1 before doing the choice $p_1A \mid p_2$ that guarantees that the PEG will backtrack to a point where p_2 matches, if possible. We can use the *FIRST* set of e_2 as an approximation to the set of possible matches of e_2 , and the PEG for $e_1^*e_2$ becomes $A \rightarrow (![FIRST(e_2)] p_1)^*(p_1A \mid p_2)$. The full rule for $\Pi(e_1^*e_2, G_k)$ becomes as follows:

$$\begin{aligned} \Pi(e_1^*e_2, G_k) &= (V_1 \cup \{A\}, T, P_1 \cup \{A \rightarrow (![FIRST(e_2)] p_1)^*(p_1A \mid p_2)\}, A) \\ &\quad \text{where } (V_1, T, P_1, p_1) = \Pi(e_1, G_2[\varepsilon]) \\ &\quad \text{with } A \notin V_1 \text{ and } G_2 = (V_2, T, P_2, p_2) = \Pi(e_2, G_k) \end{aligned}$$

Again, the easiest way to prove that this new rule is correct is by proving the equivalence of the PEGs obtained from the old and the new rule, by induction on the height of the corresponding proof trees.

4.3. Combining Search and Repetition

We can further optimize the case where we want to search for the pattern $e_1^*e_2$ or $e_1e_1^*e_2$ (we will use e_1^+ as a shorthand for $e_1e_1^*$), and all strings in $L(e_1)$

have length one. We can safely skip the prefix of the subject that matches a possessive repetition of e_1 before trying again, because if the pattern would match from any of these positions then it would not have failed in the first place. We can combine this with our first search optimization to yield the following search pattern:

$$S \rightarrow (![\mathcal{F}] \cdot)^*(p \mid p_1^*S)$$

In the pattern above, p is the starting expression of a PEG equivalent to the regular expression we are searching and p_1 is the starting expression of a PEG equivalent to e_1 with an empty continuation. Set \mathcal{F} is still the *FIRST* set of the whole regular expression. If the *FIRST* sets of e_1 and e_2 are disjoint we can further optimize our search by breaking up p and using the following search expression to search for $e_1^*e_2$:

$$S \rightarrow (![\mathcal{F}] \cdot)^*p_1^*(p_2 \mid S)$$

The special case searching for $e_1^+e_2$ just uses the search expression $S \rightarrow (![\mathcal{F}] \cdot)^*p_1^+(p_2 \mid S)$. Proofs that these optimizations are correct are straightforward, by proving that these search expressions are equivalent to $S \rightarrow (![\mathcal{F}] \cdot)^*(p \mid \cdot S)$ by induction on the height of the derivation trees.

5. Benchmarks

This section presents some benchmarks that compare a regex engine based on an implementation of our transformation with the resulting PEGs executed with LPEG, a fast backtracking PEG engine [11]. We compare this engine with PCRE, a backtracking regex engine that performs ad-hoc optimizations to cut the amount of backtracking needed [3], and with RE2, a non-backtracking (automata-based) regex engine that nevertheless also incorporates ad-hoc optimizations [8].

We tested our search and repetition optimizations with a series of benchmarks that search for the first successful match of a regular expression inside a large subject, the Project Gutenberg version of the King James Bible [18]. Our first benchmark searches for a single literal word in the subject, and serves as a simple test of the search optimization. Table 1 shows the results. We can see that the optimization is very effective, as LPEG optimizes the repetition in the search pattern to a single instruction of its parsing machine that scans the subject checking each character against a bitmap that encodes

Word	RE2	PCRE	Unoptimized	Search	Line
Geshurites	1	1	12	1	19936
worshippeth	3	3	25	4	42140
blotteth	3	3	33	6	60005
sprang	7	9	47	11	80000

Table 1: Time in milliseconds to search for a word

Words	RE2	PCRE	Unopt	Search	Repetition	Line
Adam - Eve	1	0	0	0	0	261
Israel - Samaria	2	2	32	3	3	31144
Jesus - John	2	3	73	6	6	76781
Jesus - Judas	2	4	81	6	6	84614
Jude - Jesus	3	4	94	7	7	98311
Abraham - Jesus	5	5	96	8	8	no match

Table 2: Time in milliseconds to search for two words in the same period

the character set. RE2 and PCRE use ad-hoc optimizations to find the string and are still faster in some of the cases [8].

Our second benchmark searches for two literal words in the same period (separated by letters, spaces or commas), and we test the search and repetition optimizations, but cannot apply the combined optimization of Section 4.3 because the expression does not have the necessary structure. Table 2 shows the results, and we separate the optimizations to show the contribution of each one in the final result. The runtime is still dominated by having to find where in the subject the match is, so optimizing the repetition inside the pattern does not yield any gains. The pattern starts with a literal, so RE2 and PCRE are using ad-hoc optimizations to find where the match begins.

The third benchmark searches for a literal word that follows any other word plus a single space (a regular expression $[a-zA-Z]^+ _ w$, using character class notation and $_$ for the empty space symbol). This pattern falls in the case where the *FIRST* sets of the repeated pattern and the pattern following the repetition are disjoint. We can apply the combined search and repetition optimization for this pattern, and compare it with the basic search and repetition optimizations. Table 3 shows the results. Now even the unoptimized PEG defeats a backtracking regex matcher, but the DFA-based RE2 is much faster. The *FIRST* set of the pattern includes most of terminals, and the search optimization is not effective. The biggest gain

Word	RE2	PCRE	Unopt	Search	Rep	Combined	Line
Geshurites	5	74	57	59	38	8	19995
worshippeth	7	156	121	126	86	18	42140
blotteth	10	208	159	167	121	24	60005
sprang	12	285	222	227	147	32	80000

Table 3: Time in milliseconds to search for a word following another

Words	RE2	PCRE	Unopt	Search	Rep	Comb	Line
Adam - Eve	2	4	6	6	0	0	261
Israel - Samaria	6	504	752	750	126	8	31144
Jesus - John	12	1134	1710	1718	278	18	76781
Jesus - Judas	13	1246	1884	1892	306	20	84614
Jude - Jesus	15	1446	2176	2188	364	24	98311
Abraham - Jesus	15	1470	2214	2220	362	24	no match

Table 4: Time in milliseconds to search for a period containing two words

comes from the combined optimization, as it lets the PEG skip large portions of the subject in case of failure, yielding a result that is much closer to RE2.

The fourth and final benchmark extends the second benchmark by bracketing the pattern with a pattern that matches the part of the period that precedes and follows the two words we are searching, yielding the pattern $[a - zA - Z, \lrcorner]^*w_1[a - zA - Z, \lrcorner]^*w_2[a - zA - Z, \lrcorner]^*$. There is overlap in the *FIRST* sets of $[a - zA - Z, \lrcorner]$, w_1 , and w_2 , so we need to use the more general form of the repetition optimization. We can also apply the combined optimization. As in the third benchmark, we compare this optimization with the basic search and repetition optimizations. Table 4 shows the results. The effect of the repetition optimization is bigger in this benchmark, but what brings the performance close to a DFA-based regex matcher, and much better than a backtracking regex matcher, is still the combined optimization.

Our benchmarks show that without optimizations our PEG-based engine performs on par with PCRE on more complex patterns. The optimizations bring it to a factor of 1 to 3 of the performance of RE2, a very efficient and well-tuned regex implementation that cannot implement common regex extensions due to its automata-based implementation approach.

$$\begin{aligned}
\Pi(?e_1, G_k) &= (V_1, T, P_1, p_1 p_k), \text{ where } (V_1, T, P_1, p_1) = \Pi(e_1, G_k[\varepsilon]) \\
\Pi(e_1^{*+}, G_k) &= \Pi(?e^*, G_k) \\
\Pi(e_1^{*?}, G_k) &= G, \text{ where } G = (V_1, T, P_1 \cup \{A \rightarrow p_k \mid p_1\}, A), \\
&\quad (V_1, T, P_1, p_1) = \Pi(e_1, (V_k \cup \{A\}, T, P_k, A)), \text{ and } A \notin V_k \\
\Pi(!e_1, G_k) &= (V_1, T, P_1, !p_1 p_k), \text{ where } (V_1, T, P_1, p_1) = \Pi(e_1, G_k[\varepsilon]) \\
\Pi(?!e_1, G_k) &= (V_1, T, P_1, !!p_1 p_k), \text{ where } (V_1, T, P_1, p_1) = \Pi(e_1, G_k[\varepsilon])
\end{aligned}$$

Figure 7: Adapting Function Π to Deal with Regex Extensions

6. Transforming Regex Extensions

Regexes add several ad-hoc extensions to regular expressions. We can easily adapt transformation Π to deal with some of these extensions, and this section shows how to use Π with independent expressions, possessive repetitions, lazy repetitions, and lookahead. An informal but broader discussion of regex extensions in the context of translation to PEGs was published by Oikawa et al. [19].

The regex $?e_1$ is an independent expression (also known as atomic grouping). It matches independently of the expression that follows it, so a failure when matching the expression that follows $?e_1$ does not force a backtracking regex matcher to backtrack to $?e_1$'s alternative matches. This is the same behavior as a PEG, so to transform $?e_1$ we first transform it using an empty continuation, then concatenate the result with the original continuation.

The regex e_1^{*+} is a possessive repetition. It always matches as most as possible of the input, even if this leads to a subsequent failure. It is the same as $?e^*$ if the longest-match rule is used. The semantics of Π guarantees longest match, so it uses this identity to transform e_1^{*+} .

The regex $e_1^{*?}$ is a lazy repetition. It always matches as little of the input as necessary for the rest of the expression to match (shortest match). The transformation of this regex is very similar to the transformation of e_1^* , we just flip p_1 and p_k in the production of non-terminal A . Now the PEG tries to match the rest of the expression first, and will only try another step of the repetition if the rest fails.

The regex $!e_1$ is a negative lookahead. The regex matcher tries to match the subexpression; if it fails then the negative lookahead succeeds without

consuming any input, and if the subexpression succeeds the negative lookahead fails. Negative lookahead is also an independent expression. Transforming this regex is just a matter of using PEGs negative lookahead, which works in the same way, on the result of transforming the subexpression as an independent expression.

Finally, the regex $?=e_1$ is a positive lookahead, where the regex matcher tries to match the subexpression and fails if the subexpression fails and succeeds if the subexpression succeeds, but does not consume any input. It is also an independent expression. We transform a positive lookahead by transforming the subexpression as an independent expression and then using PEGs negative lookahead twice.

None of these extensions has been formalized before, as they depend on the behavior of backtracking-based implementations of regexes instead of the semantics of regular expressions. We decided to formalize them in terms of their conversion to PEGs instead of trying to rework our semantics of regular expressions to accommodate them, as these extensions map naturally to concepts that are already part of the semantics of PEGs.

The well-formedness rewriting of Section 3.1 needs to accommodate the new extensions. It is not possible to rewrite all non-well-formed expressions with these extensions while keeping their behavior the same, as these extensions make it possible to write expressions that cannot give a meaningful result, such as $(?)(\varepsilon | a)^*$ or $(?=a(d | \varepsilon))^*$. Other expressions can work with some subjects and not work with others, such as $(?)(a | \varepsilon | b)^*$ or $(?=a(a | \varepsilon))^*$.

Our approach will be to rewrite problematic expressions so they give the same result for the subjects where they do not cause problems, but also give a result for other subjects, that is, they will match a superset of the strings that the original expression matches. For example, the four expressions above will be respectively rewritten to $(?)a^*$, d^* , $(?)(a | b)^*$, and a^* .

The *empty* predicate is **true** for $?!e_1$ and $?=e_1$ expressions, and *empty*(e_1) for the other extensions. This is a conservative definition, as expressions such as $?)(\varepsilon | e)$ can also be replaced by ε given the informal semantics of regexes. The *null* predicate is **true** for all the extensions except $?e_1$, where it is *null*(e_1).

Figure 8 gives the definitions of f_{out} and f_{in} for the extensions. Function f_{out} just applies itself recursively for $?e_1$, $?!e_1$ and $?=e_1$, but it needs to rewrite the repetitions using f_{in} if their bodies can match ε . Function f_{in} applies itself recursively to atomic groupings, keeping them atomic, but it strips repetitions. A repetition being rewritten by f_{in} is used directly inside

$$\begin{aligned}
f_{out}(?)e_1 &= ?f_{out}(e_1) \\
f_{out}(e_1^{*+}) &= \begin{cases} f_{out}(e_1)^{*+} & \text{if } \neg null(e_1) \\ \varepsilon & \text{if } empty(e_1) \\ f_{in}(e_1)^{*+} & \text{otherwise} \end{cases} \\
f_{out}(e_1^{*?}) &= \begin{cases} f_{out}(e_1)^{*?} & \text{if } \neg null(e_1) \\ \varepsilon & \text{if } empty(e_1) \\ f_{in}(e_1)^{*?} & \text{otherwise} \end{cases} \\
f_{out}(?!e_1) &= ?!f_{out}(e_1) \\
f_{out}(?=e_1) &= ?=f_{out}(e_1) \\
\\
f_{in}(?)e_1 &= ?f_{in}(e_1) \\
f_{in}(e_1^{*+}) &= \begin{cases} f_{in}(e_1) & \text{if } null(e_1) \\ f_{out}(e_1) & \text{otherwise} \end{cases} \\
f_{in}(e_1^{*?}) &= \begin{cases} f_{in}(e_1) & \text{if } null(e_1) \\ f_{out}(e_1) & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 8: Definition of Functions f_{out} and f_{in} for regex extensions

another repetition, so it does not matter if it is possessive, lazy, or a regular greedy repetition, it is the outer repetition that will govern how much of the subject will be matched.

We do not need to define f_{in} for negative and positive lookaheads, as pathological uses of these expressions are eliminated by the f_{out} case that rewrites repetitions with an *empty* body and the f_{in} cases that rewrite choices with *empty* alternatives.

The extensions do not impact the optimizations of Section 4 if we provide a way of computing a *FIRST* set for them, as the optimizations do not depend on the structure of the subexpressions they use. We obviously cannot apply the repetition optimization on $e_1^{*+}e_2$, $e_1^{*?}e_2$, or $?(e_1^*)e_2$, but applying it on $e_1^*e_2$ where extensions appear inside e_1 or e_2 is not a problem. The repetition optimizations are turning repetitions into possessive repetitions where possible, so not being able to optimize expressions such as the ones above is not a loss, as they will already exhibit good backtracking behavior.

Figure 9 gives an inductive definition for the *FIRST* sets of extended

$$\begin{aligned}
FIRST(\varepsilon) &= \emptyset \\
FIRST(a) &= \{\mathbf{a}\} \\
FIRST(e_1 e_2) &= \begin{cases} FIRST(e_1) & \text{if } \neg null(e_1) \\ FIRST(e_1) \cup FIRST(e_2) & \text{if } null(e_1) \end{cases} \\
FIRST(e_1 | e_2) &= FIRST(e_1) \cup FIRST(e_2) \\
FIRST(e^*) &= FIRST(e) \\
FIRST(?e) &= FIRST(e) \\
FIRST(e^{*+}) &= FIRST(e) \\
FIRST(e^{*?}) &= FIRST(e) \\
FIRST(!e) &= \emptyset \\
FIRST(?=e) &= \emptyset
\end{aligned}$$

Figure 9: Definition of *FIRST* sets for regexes

regexes. For completeness, we also give cases for the standard regexes. In our definition of *FIRST* sets in terms of relation $\overset{\text{RE}}{\rightsquigarrow}$ the *FIRST* sets cannot include ε , so expressions that never consume any prefix of the subject have empty *FIRST* sets. The *FIRST* sets of atomic groupings are conservative, as they may be a proper superset of the first characters that the expression actually consumes; for example, $FIRST(?)(\varepsilon | a)$ is $\{\mathbf{a}\}$ instead of the more precise \emptyset .

7. Conclusion

We presented a new formalization of regular expressions that uses natural semantics and a transformation Π that converts a given regular expression into an equivalent PEG, that is, a PEG that matches the same strings that the regular expression matches in a Perl-compatible regex implementation. If the regular expression's language has the prefix property, easily guaranteed by using an end-of-input marker, the transformation yields a PEG that recognizes the same language as the regular expression.

We also have shown how our transformation can be easily adapted to accommodate several ad-hoc extensions used by regex libraries: independent expressions, possessive and lazy repetition, and lookahead. Our trans-

formation gives a precise semantics to what were informal extensions with behavior specified in terms of how backtracking-based regex matchers are implemented.

We show that, for some classes of regular expressions, we produce PEGs that perform better by reasoning about how the PEG’s limited backtracking and syntactical predicates work to control the amount of backtracking that a PEG will perform. The same reasoning that we apply for large classes of expressions can be applied to specific ones to yield bigger performance gains where necessary, although our benchmarks show that simple optimizations are enough to perform close to optimized regex matchers, while having a much simpler implementation: both regex engines we used have over ten times the amount of code of the PEG engine.

Another approach to establish the correspondence between regular expressions and PEGs was suggested by Ierusalimsky [11]. In this approach we convert Deterministic Finite Automata (DFA) into right-linear LL(1) grammars. Medeiros [14] proves that an LL(1) grammar has the same language when interpreted as a CFG and as a PEG. But this approach cannot be used with regex extensions, as they cannot be expressed by a DFA.

The transformation II is a formalization of the continuation-based conversion presented by Oikawa et al. [19]. That work only presents an informal discussion of the correctness of the conversion, while we proved our transformation correct with regards to the semantics of regular expressions and PEGs.

We can also benefit from the LPEG parsing machine [12, 11], a virtual machine for executing PEGs. We can use the cost model of the parsing machine instructions to estimate how efficient a given regular expression or regex is. The parsing machine has a simple architecture with just nine basic instructions and four registers, and implementations of our transformation coupled with implementations of the parsing machine can be the basis for simpler implementations of regex libraries.

References

1. Wall, L.. Apocalypse 5: Pattern matching. 2002. URL <http://www.perl6.org/archive/doc/design/apo/A05.html>.
2. Conway, D., Randal, A., Michaud, P., Wall, L., Lenz, M.. Synopsis 5: Regexes and rules. 2002. URL <http://feather.perl6.nl/syn/S05.html>.

3. Hazel, P.. PCRE - Perl Compatible Regular Expressions. 2011. URL <http://www.pcre.org/>.
4. Aho, A.V.. *Algorithms for Finding Patterns in Strings*. Cambridge, MA, USA: MIT Press. ISBN 0-444-88071-2; 1990, p. 255–300.
5. Abigail. Reduction of 3-CNF-SAT to Perl regular expression matching. <http://perl.plover.com/NPC/NPC-3SAT.html>; 2001.
6. Friedl, J.. *Mastering Regular Expressions*. O'Reilly Media, Inc.; 2006. ISBN 0596528124.
7. Fowler, G.. An interpretation of the POSIX regex standard. Tech. Rep.; Information and Software Systems Research, AT&T Labs; New Jersey, USA; 2003.
8. Cox, R.. Regular expression matching in the wild. 2010. URL <http://swtch.com/~rsc/regex/regex3.html>.
9. Reps, T.. “Maximal-munch” tokenization in linear time. *ACM Transactions on Programming Languages and Systems* 1998; **20**(2):259–273. doi:\bibinfo{doi}{10.1145/276393.276394}. URL <http://doi.acm.org/10.1145/276393.276394>.
10. Ford, B.. Parsing Expression Grammars: A recognition-based syntactic foundation. In: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, USA: ACM. ISBN 1-58113-729-X; 2004, p. 111–122.
11. Ierusalimschy, R.. A text pattern-matching tool based on Parsing Expression Grammars. *Software - Practice & Experience* 2009;**39**(3):221–258.
12. Medeiros, S., Ierusalimschy, R.. A parsing machine for PEGs. In: *DLS '08: Proceedings of the 3rd Dynamic Languages Symposium*. New York, USA: ACM. ISBN 978-1-60558-270-2; 2008, p. 1–12.
13. Kahn, G.. Natural semantics. In: *STACS '87: Proceedings of the 4th Symposium on Theoretical Aspects of Computer Science*. London, UK: Springer-Verlag. ISBN 3-540-17219-X; 1987, p. 22–39.

14. Medeiros, S.. *Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto*. Ph.D. thesis; PUC-Rio; 2010.
15. Hopcroft, J., Ullman, J.. *Introduction to Automata Theory, Languages, and Computation*. Boston, USA: Addison-Wesley Longman Publishing Co., Inc.; 1979. ISBN 0321462254.
16. Salomaa, A.. Two complete axiom systems for the algebra of regular events. *Journal of the ACM* 1966;**13**(1):158–169. doi:\bibinfo{doi}{10.1145/321312.321326}. URL <http://doi.acm.org/10.1145/321312.321326>.
17. The Perl Foundation. Core documentation for Perl 5 version 14.1 – regular expressions. <http://perldoc.perl.org/perlre.html>; 2005.
18. Anonymous. *The Bible, Both Testaments, King James Version*; vol. 10. P.O. Box 2782, Champaign, IL 61825-2782, USA: Project Gutenberg; 1989.
19. Oikawa, M., Ierusalimsky, R., Moura, A.. Converting regexes to Parsing Expression Grammars. In: *SBLP '10: Proceedings of the 14th Brazilian Symposium on Programming Languages*. 2010, .